

Fast Classification and Clustering via Image Convolution Filters

Alternative to Generative Mixture Models

Vincent Granville, Ph.D.
vincentg@MLTechniques.com
www.MLTechniques.com
Version 1.0, June 2022

Abstract

I generate synthetic data using a superimposition of stochastic processes, comparing it to Bayesian generative mixture models (Gaussian mixtures). I explain the benefits and differences. The actual classification and clustering algorithms are model-free, and performed in GPU as image filters, after transforming the raw data into an image. I then discuss the generalization to 3D or 4D, and to higher dimensions with sparse tensors. The technique is particularly suitable when the number of observations is large, and the overlap between clusters is substantial.

It can be done using few iterations and a large filter window, comparable to a neural network, with pixels in the local window being the nodes, and their distance to the local center being the weight function. Or you can implement the method with a large number of iterations – the equivalent of hundreds of layers in a deep neural network – and a tiny window. This latter case corresponds to a sparse network with zero or one connection per node. It is used to implement fractal classification, where point labeling changes at each iteration, around highly non-linear cluster boundaries. This is equivalent to putting a prior on class assignment probabilities in a Bayesian framework. Yet, classification is performed without underlying model. Finally, the clustering (unsupervised) part of the algorithm relies on the same filtering techniques, combined with a color equalizer. The latter can be used to perform hierarchical clustering.

The Python code, included in this document, is also on my GitHub repository. A data animation illustrates how simple the methodology is: each frame in the video represents one iteration, that is, a single application of the filter to all the data points. Indeed, the classifier can be used as a black box system. It follows the modern trend of interpretable machine learning, also called explainable AI. The video shows how the algorithm converges to an optimum, producing a classification of the entire observation space. Classifying a new point is then immediate: read its color. The whole system is time-efficient. It does not require the computation of all training set point intra-distances. However it is memory-intensive. Large filters can be slow, though they require very few iterations. I discuss a simple technique to make them a lot faster.

Contents

1	Introduction	1
2	Generating the synthetic data	2
2.1	Simulations with logistic distribution	2
2.2	Mapping the raw observations onto an image bitmap	3
3	Classification and unsupervised clustering	3
3.1	Supervised classification based on convolution filters	4
3.2	Clustering based on histogram equalization	4
3.3	Fractal classification: deep neural network analogy	5
3.4	Generalization to higher dimensions	6
3.5	Towards a very fast implementation	7
4	Python code	7
4.1	Fractal classification	8
4.2	GPU classification and clustering	10
4.3	Home-made graphic library	12
	References	15

1 Introduction

I explain, with Python code and numerous illustrations, how to turn traditional tabular data into images, to perform both clustering and supervised classification using simple image filtering techniques. I also explain

how to generalize the methodology to higher dimensions, using tensors rather than images. In the end, image bitmaps are 2D arrays or matrices, that is, 2D tensors. By classifying the entire space (in low dimensions), the resulting classification rule is very fast. I also discuss the convergence of the algorithm, and how to further improve its speed.

This short article covers many topics and can be used as a first introduction to synthetic data generation, mixture models, boundary effects, explainable AI, fractal classification, stochastic convergence, GPU machine learning, deep neural networks, and model-free Bayesian classification. I use very little math, making it accessible to the layman, and certainly, to non-mathematicians. Introducing an original, intuitive approach to general classification problems, I explain in simple English how it relates to deep and very deep neural networks. In the process, I make connections to image segmentation, histogram equalization, hierarchical clustering, convolution filters, and stochastic processes. I also compare standard neural networks with very deep but sparse ones, in terms of speed and performance. The fractal classifier – an example of very deep neural network – is illustrated with a Python-generated video. It is useful when dealing with massively overlapping clusters and a large number of observations. Hyperparameters allow you to fine tune the level of cluster overlap in the synthetic data, and the shape of the clusters.

2 Generating the synthetic data

The data used in my examples is generated using a technique that bears some resemblance to [Gaussian mixture models](#) (GMM) in a Bayesian framework [\[Wiki\]](#). Instead of mixtures, I used superimposed stochastic processes, also called interlacings. The differences with mixtures are subtle and can be detected with statistical tests, but unimportant and not visible to the naked eye. And rather than Gaussian distributions, I use arbitrary distributions. The mathematical model is that of stochastically perturbed lattice processes, also known as Poisson-binomial point processes.

I wrote a comprehensive 100-page book on the topic [\[2\]](#), with numerous references, covering all aspects from simulation, graph properties, to theory. These processes have become popular recently, with applications to sensor data, cell networks, crystallography and chemistry. They are flexible and very easy to simulate. The three main parameters are the scale or diffusion factor s , the intensity λ (linked to the expected number of points per unit area), and the local distribution F which may be Gaussian or not. In this article, I don't discuss the theory. I only introduce the basic material necessary to generate the synthetic data. The reader is referred to my book [\[2\]](#) for details.

First, I use $\lambda = 1$ in the Python code included in this document. Then, if $s = 0$, the points are all located on a lattice: there is no randomness anymore. To the contrary, if s is large (say $s > 5$) then the points are nearly uniformly distributed, as in a Poisson point process. In that case, the simulated data has no clustering structure. The synthetic data here uses either $s = 0.05$ resulting in well separated clusters, or $s = 0.15$ resulting in significant cluster overlap.

For examples with five clusters, see left plot in [Figure 3](#) (with $s = 0.15$) and [Figure 4](#) (with $s = 0.05$). For four clusters, see left plot in [Figure 5](#) (with $s = 0.15$) and [Figure 6](#) (with $s = 0.05$). The number of clusters, denoted as m , is the number of components (stochastic processes) used in the superimposition.

Thus, the cluster structure is generated by interlacing multiple stretched and shifted perturbed lattices. The stretching factor and intensity may be different depending on the direction. A special case with $s = 0$ is the deterministic hexagonal lattice pictured in [Figure 1](#). Numerous examples with various degrees of randomness are pictured in my book [\[2\]](#). If $s > 0$, the points of a single process are independently distributed with multivariate distribution F , around each lattice vertex. If $s = 0$, the data points are the lattice vertices, as in [Figure 1](#).

2.1 Simulations with logistic distribution

All simulations are in two dimensions. There is no particular reason to choose a logistic distribution for F , other than that it is the easiest to sample from. A Gaussian distribution would work too. The distribution F has little impact on the final results. Indeed, it is not easy and sometimes impossible to reverse-engineer the system to identify the underlying distribution F . See my book [\[2\]](#) page 33 for a discussion on this topic. Finally, the algorithm is as follows:

Data generation: algorithm

The data generated is 2D, but it is easy to generalize to any dimension. For each lattice vertex (h, k) , where h, k are integers (positive or negative), and for each stochastic lattice process M_i with $0 \leq i < m$, generate the bivariate observation (x_{ih}, y_{ik}) as follows:

$$x_{ih} = \mu_i + \frac{h}{\lambda_i} + s \cdot \log \left(\frac{U_{ih}}{1 - U_{ih}} \right) \quad (1)$$

$$y_{ik} = \mu'_i + \frac{k}{\lambda'_i} + s \cdot \log \left(\frac{U_{ik}}{1 - U_{ik}} \right) \quad (2)$$

Formulas (1) and (2) are from my book [2], page 11. The U_{ih}, U_{ik} are independent uniform deviates on $[0, 1]$. In practice, we generate points in a finite rectangular window, and we only keep those inside a sub-window, to avoid the boundary effects described in my book. This is implemented in the Python code in section 4, with $-25 \leq h, k \leq 25$. In the code, I use the arrays `stretchX`, `stretchY` to store the coefficients $1/\lambda_i, 1/\lambda'_i$, and `shiftX`, `shiftY` to store the shift vectors (μ_i, μ'_i) . The shift vectors are the centers of the clusters. In the rectangular window, by design, the number of observed points from the process M_i (which plays the role of a mixture component in a mixture model) is proportional to $\lambda_i \times \lambda'_i$ in two dimensions. In my examples, I chose $\lambda_i = \lambda'_i = \lambda$, with $\lambda = 1$.

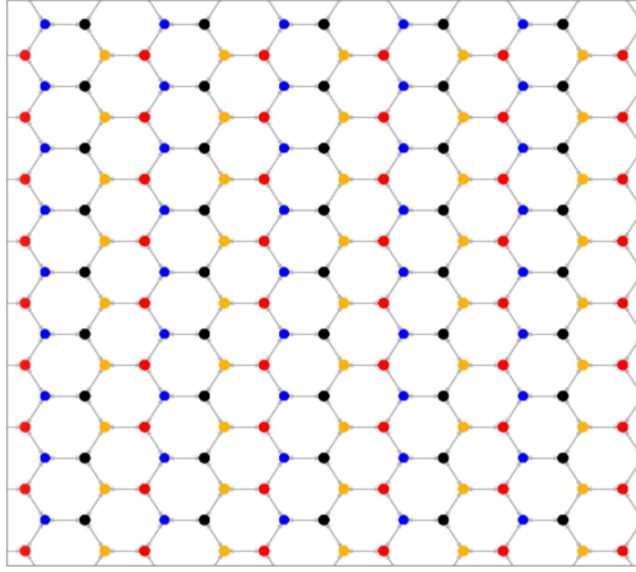


Figure 1: Special interlacing of 4 lattice processes with $s = 0$.

2.2 Mapping the raw observations onto an image bitmap

Eventually, due to the lattice nature of the stochastic processes involved (with point patterns exhibiting statistical tiling around each vertex), the points are transformed using a modulo operator (see `xmod` and `ymod` in the Python code) and then mapped onto an image bitmap (the bivariate array `bitmap` in the Python code). From there, image processing techniques are used to perform classification or clustering.

3 Classification and unsupervised clustering

I describe here a methodology for fast supervised and unsupervised classification. The data is first transformed into a two-dimensional array called *bitmap*. The points are referred to as pixels, and the array represents an image stored in GPU (the graphics processing unit) [Wiki]. The functions applied to the bitmap are standard image processing techniques such as high pass filtering or histogram equalization [Wiki].

The input data consists of a realization (obtained by simulation in section 2.1 and 2.2) of an *m*-interlacing (that is, a superimposition of m shifted Poisson-binomial processes) with each individual process represented by a different color: see Figure 2 and 3. The left plot shows the data points observed modulo $2/\lambda$. So, the point locations, after the modulo operation, are in $[0, 2/\lambda[\times [0, 2/\lambda[$. I chose $\lambda = 1$ for the intensity function, in the simulations.

The modulo operator magnifies the cluster structure, which is otherwise invisible to the naked eye. It is defined as $a \bmod b = a - b[a/b]$ where the brackets represent the integer part function. For your own simulations, you can use modulo $1/\lambda$, rather than $2/\lambda$: this will remove the apparent stochastic duplication in my pictures. The reason I chose $2/\lambda$ is due to boundary effects, with clusters extending beyond the window of observations and truncated because of the window, thus making the cluster structure much harder to see if using modulo

$1/\lambda$. For the mathematically inclined reader, the methodology performs [classification on the torus](#) [Wiki] rather than on the plane: this is a standard technique when facing boundary effects. If all your data fits nicely in the observation window, you can ignore the modulo transformation.

The middle and right plots in Figure 3 correspond to [unsupervised clustering](#). The centers of the darkest areas provide an approximation to the unknown shift vectors (μ_i, μ'_i) of formulas (1) and (2), with $i = 0, \dots, m$ indicating the (unknown) cluster label. The shift vectors are the theoretical cluster centers. The approximation is far from perfect due to massive cluster overlapping. The situation is much better in Figure 4 (right plot), where cluster overlapping is much less pronounced. The methodology is described in section 3.1.

The middle and right plots in Figure 2 correspond to [supervised classification](#) of the entire space: the color of a point represents the individual point process or cluster it belongs to. In this case the data set is the training set. The methodology is described in section 3.2.

3.1 Supervised classification based on convolution filters

Here the synthetic dataset represents the training set. The algorithm consists of filtering the whole bitmap N times. Each time, a local filter is applied to each pixel (x, y) . Initially, the color $c(x, y)$ attached to the pixel represents the cluster it belongs to, in the training set (or in other words, the individual point process it originates from in the m -mixture): its value is an integer between 0 and $m - 1$ if it is in the training set, and 255 otherwise. The new color assigned to (x, y) is

$$c'(x, y) = \arg \max_j \sum_{u=-w}^w \sum_{v=-w}^w \frac{\chi[c(x-u, y-v) = j]}{\sqrt{1+u^2+v^2}}, \quad (3)$$

that is, the value of j that maximizes (3). Here $\chi[A]$ is the indicator function [Wiki]: $\chi[A] = 1$ if A is true, and 0 otherwise. The boundary problem (when $x - u$ or $y - v$ is outside the bitmap) is handled in the source code.

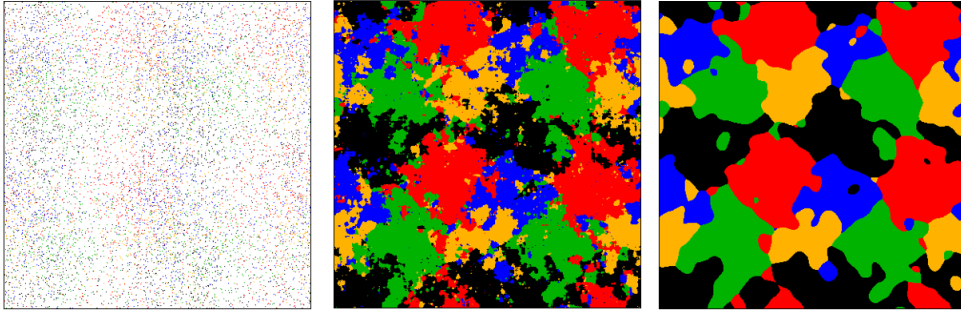


Figure 2: Classification of left dataset; $s = 0.15$, $w = 10$. One loop (middle) vs 3 (right).

In the python code, N is the parameter `nloop`, and w is the parameter window. It is also referred to as w and *loops* in the figures. In particular, I used $N = 3$ and $w = 20$. Formula (3) corresponds to `method = 1` in the Python code. While slow, it provides granular cluster boundaries. A faster version, namely `method = 0`, does not make the division by $\sqrt{1+u^2+v^2}$. It is faster not only because it avoids the square root computations, but also because it can be implemented very efficiently: see section 3.5. Yet, the loss of accuracy when using the fast method, while noticeable, is smaller than expected. See Figure 7 for comparisons.

Note that each cluster, even when the overlap is small, extends to the entire plane. So there is always some degree of overlap. But the overlap is much smaller when the diffusion factor s (the variable s in the Python code) is small. This is evident when comparing Figure 2 with Figure 4. Both figures have the same number of clusters, and the same cluster centers; only s – and thus the amount of cluster overlap – is different.

After filtering the whole bitmap $N = 3$ times, thanks to the large size of the local filtering window ($w = 20$), all pixels are assigned to a cluster. This means that any future point (not in the training set) can easily and efficiently be classified: first, find its location on the bitmap; then its cluster is the color assigned to that location. It is worth asking whether convergence occurs (and to what solution) if you were to filter the bitmap many times. I studied convergence for a similar type of filter, in my paper “Simulated Annealing: A Proof of Convergence” [3]. Empirical evidence suggests that additional loops (increasing N beyond $N = 3$) barely makes any difference.

3.2 Clustering based on histogram equalization

I use the same filter for unsupervised clustering. Indeed, both supervised and unsupervised clustering are implemented in parallel in the source code, within the same loop. The main difference is that the color (or

cluster) $c(x, y)$ attached to a pixel (x, y) is not known. Instead of colors, I use gray levels representing the density of points at any location on the bitmap: the darker, the higher the density. I start with a bitmap where $c(x, y) = 1$ if (x, y) corresponds to the location of an observed point on the bitmap, and $c(x, y) = 0$ otherwise. Again, I filter the whole bitmap $N = 3$ times with the same filter size $w = 20$. The new gray level assigned to pixel (x, y) at loop t is now

$$c'(x, y) = \arg \max_j \sum_{u=-w}^w \sum_{v=-w}^w \frac{c(x-u, y-v) \cdot 10^{-t}}{\sqrt{1+u^2+v^2}}. \quad (4)$$

The first time this filter is applied to the whole bitmap, I use $t = 0$ in Formula (4); the second time I use $t = 1$, and the third time I use $t = 2$. The purpose is to dampen the effect of successive filtering, otherwise the image (rightmost plots in Figure 3) would turn almost black everywhere after a few loops, making it impossible to visualize the cluster structure. The second and third loops, with the damping factor, provide an improvement over using a single loop only.

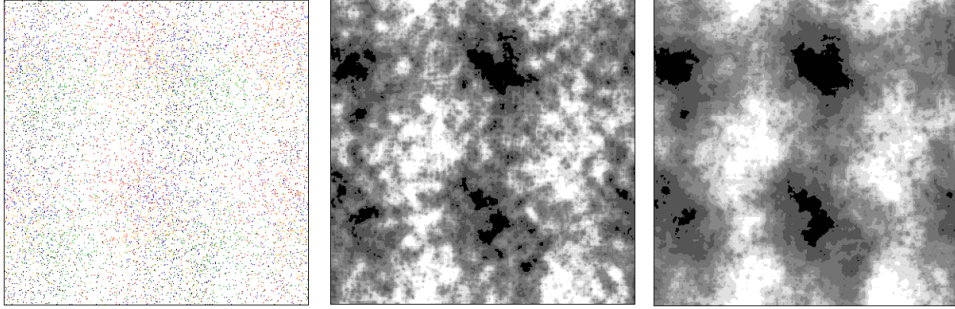


Figure 3: Clustering of left dataset; $s = 0.15$, 3 loops, $w = 10$ (middle) vs 20 (right).

After filtering the image, I use a final post-processing step to enhance the gray levels: see Part 4 of the source code in the `GD_maps` function. It consists of binning and rescaling the histogram of gray levels to make the image sharper and easier to interpret, with 8 gray levels only. This step, called [histogram equalization](#), can be automated. The successive gray levels, starting with the darkest one, correspond to successive levels in an [hierarchical clustering](#) algorithm [Wiki]. To finalize the clustering procedure, one may use [image segmentation](#) techniques [Wiki] to identify the boundary of the clusters.

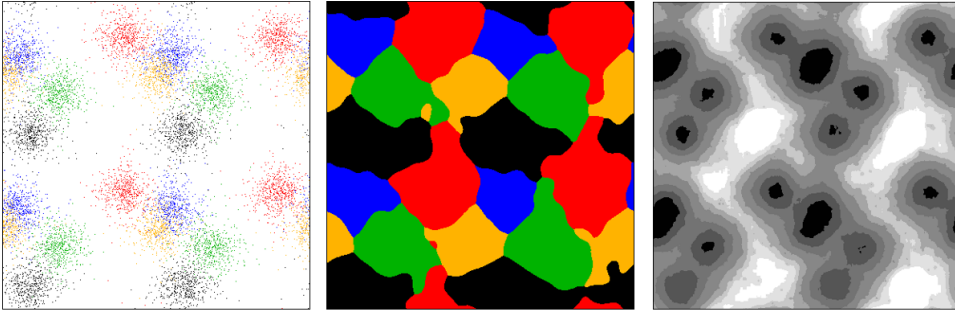


Figure 4: Classification ($w = 10$) and clustering ($w = 20$); $s = 0.05$, three loops.

The equalizer used in my code works on all the images tested. However, you may want to use one that is image-specific. The Python pillow library offers an easy way to do it, see [here](#). You can also write your own Python code for full control. See the histogram equalization code in the gigantic Algorithms repository on GitHub, [here](#).

3.3 Fractal classification: deep neural network analogy

The filtering system is essentially a [neural network](#) [Wiki]. The image before the first loop (Figures 2 and 4, left plot), consisting of the training set, is the input layer. The final image obtained after 3 loops is the output layer. The intermediate iterations correspond to the hidden layers. In each layer, the pixel color is a function of quantities computed on neighboring pixels, in the previous layer. The pre-processing step consists of transforming the data set into an image bitmap. In section 3.2 about unsupervised clustering, the post-processing step called “equalizer” plays the role of the sigmoid function in neural networks. See Luuk

Spreeuwers’ PhD thesis “Image Filtering with Neural Networks” defended in 1992 [4] (available online, [here](#)), for more about image filters used as neural networks.

In my video posted [here](#) (YouTube) and [here](#) (animated gif on GitHub), each frame represents a layer in a very deep neural network. In my methodology, I use the term “loop” or “iteration” instead of layer. It is represented by the Python variable `loop` in `PB_clustering_video.py` (section 4.1). A pixel plays the role of a neuron, and the weight attached to the link between a pixel and one of its neighbors – as in formula (3) – is also called “weight”, or parameter, in neural network terminology.

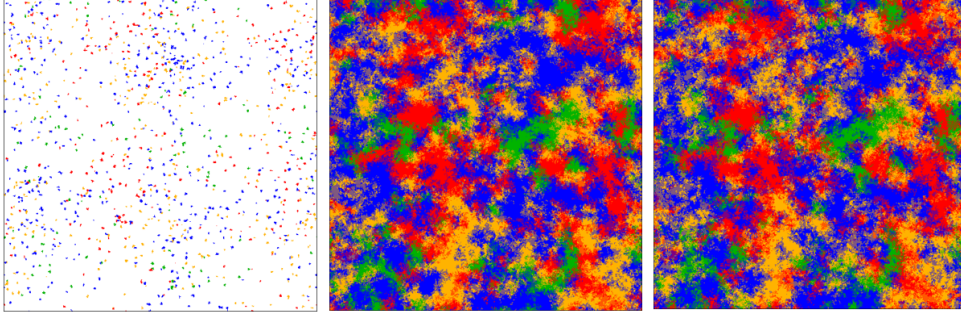


Figure 5: Fractal classification, $s = 0.15$. Loop 6, 250 and 400.

The fractal classifier described here, displayed in the video and also pictured in Figures 5 and 6, is in some sense the opposite of the one described in section 3.1: instead of using $N = 3$ loops (that is, 3 layers), it uses hundreds of them. But the local filter is extremely small, with $w \leq 1$, compared to $w = 20$ in section 3.1. Thus, each neuron (pixel) is connected to one neuron at most. Such neural networks are called “sparse”. In the end, it produces similar results, compared to using few loops and a large local filtering window. The main difference is that cluster boundaries are less smooth, and appear fractal-like. The video ([here](#)) shows the successive image transformations taking place from one loop to the next one. By watching it, it is very easy to understand how the method works, making it a classic example of [explainable AI](#) [Wiki].

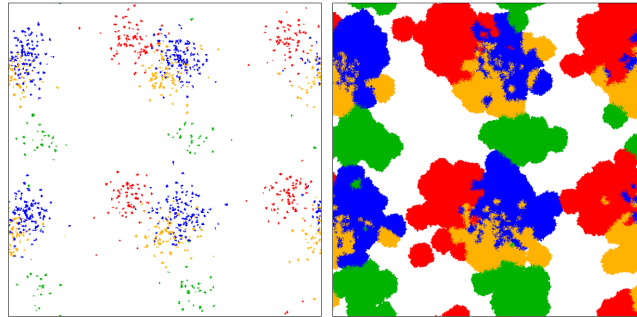


Figure 6: Fractal classification, $s = 0.05$. Loop: 6 and 60.

Once the whole state is classified (when no white area is left on the image), each subsequent loop randomly re-assigns the pixel labels (the cluster they belong to), around cluster borders. It allows you to compute, for a pixel on the border between two or more clusters, the a-posteriori probability that it belongs to any of these clusters. This makes the methodology similar to a [Bayesian classifier](#) [Wiki]. The borders between clusters are statistically stable over time: the algorithm converges, at least from a stochastic point of view.

3.4 Generalization to higher dimensions

All the examples featured in this article are in two dimensions. This makes it easy to use image processing techniques for classification or clustering. In three dimensions, images can be replaced by videos, and one can still use standard filtering techniques. It becomes more challenging in four dimensions. One way to handle the problem in higher dimensions (or even in two dimensions) is to use [tensors](#) [Wiki]. A 2D tensor is a standard rectangular matrix. A 3D tensor is a $p \times q \times r$ matrix, or in other words, a cubic matrix. Each “slice” of a 3D tensor can be treated as an image. The filters can be adapted to this type of data.

In higher dimensions, the training set occupies a tiny portion of the whole space. It does not make sense to try to classify the whole space: this becomes time-prohibitive in dimension 5 and above. Instead, the solution consists of working with [sparse tensors](#). They can be represented as graph structures, with each point connected

to its neighbors, then the neighbors of the neighbors and so on, with a depth of 4 or 5 levels. This is still a work in progress.

3.5 Towards a very fast implementation

The size w of the local filter window is the bottleneck. When filtering the image using the algorithm in section 3.1, the window used at (x, y) , and the next one at $(x + 1, y)$, both have $(2w + 1)^2$ pixels, but these two windows have $(2w + 1)^2 - 2 \times (2w + 1)$ pixels in common. So rather than visiting $(2w + 1)^2$ pixels each time, the overlapping pixels can be kept in a $(2w + 1)^2$ buffer. To update the buffer after visiting a pixel and moving to the next one to the right, one only has to update $2w + 1$ values in the buffer: overwrite the column corresponding to the old $2w + 1$ leftmost pixels, by the values derived from the new $2w + 1$ rightmost pixels.

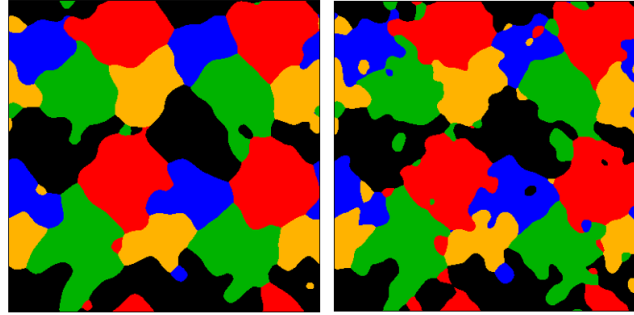


Figure 7: Fast (left) vs standard method (right), 3 loops, $s = 0.15, w = 10$.

This leads to a particularly efficient implementation when using `method = 0` (the fast filter). Then, the `GD_Maps` function in `GD_util.py` can be further optimized, since it only counts pixels (based on their color) in the local filter window, without computing distances to the center of the window. It will speed up the procedure by a factor proportional to w , both for supervised classification and clustering. Since I use $w = 20$ (the parameter window in the code), the improvement is significant.

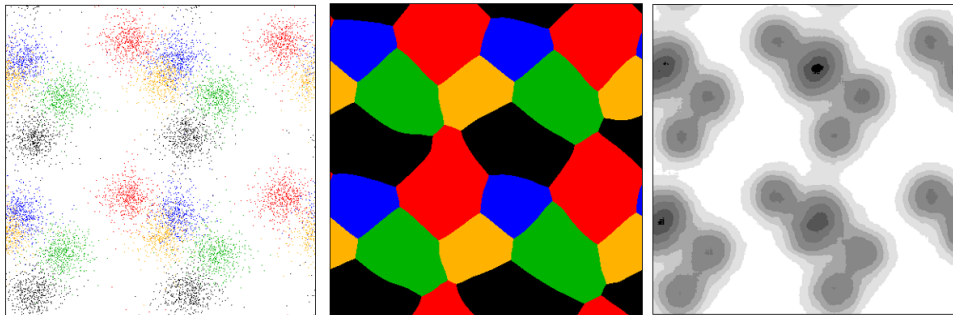


Figure 8: Fast method, $s = 0.05, w = 20$. Three loops (middle), one loop (right).

4 Python code

The Python code uses the `pillow` and `moviepy` libraries. To install Pillow, type `pip install pillow` on the Windows command prompt. The program `PB_clustering_video.py` in section 4.1 is a self-contained and short, separate piece of code. It is also the only one that produces videos (MP4 files). You might want to look at it first. The parameter s , represented by the global variable `s`, is called the scaling or diffusion factor. It determines the amount of overlap between clusters. If $s = 0$, all the points are located on a lattice. If s is large (say $s > 5$) the points are almost uniformly distributed; clustering becomes meaningless, no matter what algorithm you use.

The main program `PB_NN.py` in section 4.2 does not include any video / image processing. However, it requires `GD_util.py` (see section 4.3), my home-made small library consisting of one function `GD_Maps`. All the image processing is performed in that function. The `GD_util.py` file is assumed to be in the same folder as `PB_NN.py`. Part 3 and 4 in `PB_NN.py` deal with the time-intensive computation of all intra-distances. You don't need it, and it is turned off by the global variable `NNflag`, set to `False`. It is provided only for compatibility with an older Perl version in the first edition of my book on stochastic processes [2]. This Python version will replace the Perl code, in the new edition.

The output of `PB_NN.py` consists of PNG images, one for classification and one for clustering, per iteration. The input is the synthetic data created in part 2, and transformed into a bitmap (array of colored pixels) also in part 2. Image filtering, and thus classification and clustering, takes place in part 5: it consists of a call to the `GD_maps` function. As discussed earlier, parts 3 and 4 are skipped. The hyperparameter window, referred to as w in the figure captions, is the size of the local filter. To assign a cluster to a point (that is, a color to a pixel), the local filter consists of a $(2w + 1) \times (2w + 1)$ window centered at the point in question. Iterations (or layers, if it was a neural network) are referred to as “loops” in figure captions. The number of iterations is determined by the variable `nloop` in the code. Finally, `Nprocess` is the number of clusters, and `method` determines the type of filter. The methodology has been extensively tested with `method = 1`, which is time consuming if w is large. Note that `method = 0` still provides satisfactory results, and can be implemented in a very efficient way, as discussed in section 3.5.

For colors, I use the `RGBA` model [Wiki]. However, for the time being, the `A` component or fourth element of the color vector, known as transparency level, is not used. A future version of the code may use it, in a way similar to the supervised classification technique described in my article on visualizing high dimensional data [1].

4.1 Fractal classification

On GitHub: [PB_clustering_video.py](#). Produces the fractal classification video with 400 frames or layers, using the smallest possible filter window. Short, self-sufficient code, using the `Pillow` and `Moviepy` libraries. The input data is synthetic and created in the code: it shares some features with Bayesian Gaussian mixtures. However the classification itself is model-free. Description in section 3.3.

```
# PB_clustering_video.py

import math
import random
from PIL import Image, ImageDraw # ImageDraw to draw rectangles etc.
import moviepy.video.io.ImageSequenceClip # to produce mp4 video

Nprocess=4 # number of processes in the process superimposition
seed=82431 # arbitrary number
random.seed(seed) # initialize random generator
s=0.05 # scaling factor
shiftX=[]
shiftY=[]

for i in range(Nprocess) :
    shiftX.append(random.random())
    shiftY.append(random.random())
processID=0
height,width = (800, 800)
bitmap = [[255 for k in range(height)] for h in range(width)]

for h in range(-25,26):
    for k in range(-25,26):
        for processID in range(Nprocess):
            ranx=random.random()
            rany=random.random()
            ranID=random.random()
            if ranID < 0.20:
                processID=0
            elif ranID < 0.60:
                processID=1
            elif ranID < 0.90:
                processID=2
            else:
                processID=3
            x=shiftX[processID]+h+s*math.log(ranx/(1-ranx))
            y=shiftY[processID]+k+s*math.log(rany/(1-rany))
            if x>-3 and x<3 and x>-3 and x<3:
                xmod=1+x-int(x) # x modulo 2/lambda
                ymod=1+y-int(y) # y modulo 2/lambda
                pixelX=int(width*xmod/2)
```

```

        pixelY=int(height*(2-ymod)/2) # pixel (0,0) at top left corner
        bitmap[pixelX][pixelY]=processID

#---
img1 = Image.new( mode = "RGBA", size = (width, height), color = (0, 0, 0) )
pix1 = img1.load() # pix[x,y]=col[n] to modify the RGB color of a pixel
draw1 = ImageDraw.Draw(img1, "RGBA")

coll=[]
coll.append((255,0,0,255))
coll.append((0,0,255,255))
coll.append((255,179,0,255))
coll.append((0,179,0,255))
coll.append((0,0,0,255))
for i in range(Nprocess,256):
    coll.append((0,0,0,255))

for pixelX in range(0,width):
    for pixelY in range(0,height):
        topProcessID=bitmap[pixelX][pixelY]
        pix1[pixelX,pixelY]=coll[topProcessID]

draw1.rectangle((0,0,width-1,height-1), outline ="black",width=1)
fname="img_0.png"
img1.save(fname)

#---
nloop=400 # number of times the image is filtered

oldBitmap = [[255 for k in range(height)] for h in range(width)]
flist=[]

for loop in range(1,nloop+1):
    print("loop", loop, "out of", nloop+1)
    for pixelX in range(0,width):
        for pixelY in range(0,height):
            oldBitmap[pixelX][pixelY]=bitmap[pixelX][pixelY]
    for pixelX in range(1,width-1):
        for pixelY in range(1,height-1):
            x=pixelX
            y=pixelY
            topProcessID=oldBitmap[x][y]
            if topProcessID==255 or loop>50:
                r=random.random()
                if r<0.25:
                    x=x+1
                    if x>width-2:
                        x=x-(width-2)
                elif r<0.5:
                    x=x-1
                    if x<1:
                        x=x+width-2
                elif r<0.75:
                    y=y+1
                    if y>height-2:
                        y=y-(height-2)
                else:
                    y=y-1
                    if y<1:
                        y=y+height-2
            if loop>=50 and oldBitmap[x][y]==255:
                x=pixelX
                y=pixelY
            topProcessID=oldBitmap[x][y]
            bitmap[pixelX][pixelY]=topProcessID
            pix1[pixelX,pixelY]=coll[topProcessID]

```

```

draw1.rectangle((0,0,width-1,height-1), outline ="black",width=1)
fname="img_"+str(loop+1)+'.png'
flist.append(fname)
img1.save(fname)

clip = moviepy.video.io.ImageSequenceClip.ImageSequenceClip(flist, fps=20)
clip.write_videofile('img.mp4')

```

4.2 GPU classification and clustering

On GitHub: [PB_NN.py](#). Produces the supervised classification and clustering using a large filter window and only 3 layers. Requires the small, home-made graphic library `GD_util.py` featured in section 4.3. All image manipulations are performed in that library. The input data is synthetic and created in the code: it shares some features with Bayesian Gaussian mixtures. However the classification itself is model-free. Description in sections 3.1 and 3.2.

```

# PB_NN.py
# lambda = 1

import numpy as np
import math
import random

#---
# PART 1: Initialization

Nprocess=5          # number of processes in the process superimposition
s=0.15              # scaling factor
method=1            # method=0 is fastest
NNflag=False        # set to True if you need to compute NN distances
window=20           # determines size of local filter [the bigger, the smoother]
nloop=3             # number of times the image is filtered [the bigger, the smoother]

epsilon=0.0000000001 # for numerical stability
seed=82431          # arbitrary number
random.seed(seed)   # initialize random generator

sep="\t"  # TAB character
shiftX=[]
shiftY=[]
stretchX=[]
stretchY=[]
a=[]
b=[]
process=[]
sstring=[] # string in Perl version

for i in range(Nprocess) :
    shiftX.append(random.random())
    shiftY.append(random.random())
    stretchX.append(1.0)
    stretchY.append(1.0)
    sstring.append(sep)
    # i TABs separating x and y coordinates in output file for points
    # originating from process i; Used to easily create a scatterplot in Excel
    # with a different color for each process.
    sep=sep + "\t"

processID=0
m=0 # number of points generated
height,width = (400, 400)

bitmap = [[255 for k in range(height)] for h in range(width)]

#---

```

```

# PART 2: Generate point process, its modulo 2 version; save to bitmap and output files.

OUT = open("PB_NN.txt", "w") # the points of the process
OUT2 = open("PB_NN_modulo.txt", "w") # the same points modulo 2/lambda both in x and y
    directions

for h in range(-25,26):
    for k in range(-25,26):
        for processID in range(Nprocess):
            ranx=random.random()
            rany=random.random()
            x=shiftX[processID]+stretchX[processID]*h+s*math.log(ranx/(1-ranx))
            y=shiftY[processID]+stretchY[processID]*k+s*math.log(rany/(1-rany))
            a.append(x) # x coordinate attached to point m
            b.append(y) # y coordinate attached to point m
            process.append(processID) # processID attached to point m
            m=m+1
            line=str(processID)+"\t"+str(h)+"\t"+str(k)+"\t"+str(x)+sstring[processID]+str(y)+"\n"
            OUT.write(line)
            # replace sstring[processID] by \t if you don't care about Excel

            if x>-20 and x<20 and y>-20 and y<20:
                xmod=1+x-int(x) # x modulo 2/lambda
                ymod=1+y-int(y) # y modulo 2/lambda
                pixelX=int(width*xmod/2)
                pixelY=int(height*(2-ymod)/2) # pixel (0,0) at top left corner
                bitmap[pixelX][pixelY]=processID
                line=str(xmod)+sstring[processID]+str(ymod)+"\n"
                OUT2.write(line)
                # replace sstring[processID] by \t if you don't care about Excel

OUT2.close()
OUT.close()

#---
# PART 3: Find nearest neighbor points, and compute nearest neighbor distances.

if NNflag:

    OUT = open("PB_NN_dist_small.txt", "w") # the points of the process
    OUTf = open("PB_NN_dist_full.txt", "w") # the same points modulo 2/lambda both in x and
        y directions

    NNx=[]
    NNy=[]
    NNidx=[]
    NNidxHash={}

    for i in range(m):
        NNx.append(0.0)
        NNy.append(0.0)
        NNidx.append(-1)
        mindist=99999999
        flag=-1
        if a[i]>-20 and a[i]<20 and b[i]>-20 and b[i]<20:
            flag=0;
            for j in range(m):
                dist=math.sqrt((a[i]-a[j])**2 + (b[i]-b[j])**2) # taxicab distance faster to
                    compute
                if dist<=mindist+epsilon and i!=j:
                    NNx[i]=a[j] # x-coordinate of nearest neighbor of point i
                    NNy[i]=b[j] # y-coordinate of nearest neighbor of point i
                    NNidx[i]=j # indicates that point j is nearest neighbor to point i
                    # NNidxHash[i] is the list of points having point i as nearest neighbor;
                    # these points are separated by "~" (usually only one point in NNidxHash[i]
                    # unless the simulated points are exactly on a lattice, e.g. if s = 0)
                    if abs(dist-mindist) < epsilon:

```

```

        NNidxHash[i]=NNidxHash[i]+"~"+str(j)
    else:
        NNidxHash[i]=str(j)
    mindist=dist
    if i % 100 == 0:
        print("Finding Nearest neighbors of point",i)
    line=str(i)+"\t"+str(mindist)+"\n"
    OUT.write(line)
    line=str(i)+"\t"+str(NNidx[i])+"\t"+str(NNidxHash[i])+"\t"+str(a[i])+"\t"
    line=line+str(b[i])+"\t"+str(NNx[i])+"\t"+str(NNy[i])+"\t"+str(mindist)+"\n"
    OUTf.write(line)

OUTf.close()
OUT.close()

#---
# PART 4: Produce data to use in R code that generates the nearest neighbors picture.

if NNflag:

    OUT = open("PB_r.txt","w")
    OUT.write("idx\tNN\tNNindex\ta\tb\tANN\tbNN\tprocessID\tNNprocessID\n")

    for idx in NNidxHash:
        NNlist=NNidxHash[idx]
        list=NNlist.split("~")
        nelts=len(list)
        for n in range(nelts):
            NNindex=int(list[n])
            line=str(idx)+"\t"+str(n)+"\t"+str(NNindex)+"\t"+str(a[idx])+"\t"+str(b[idx])
            line=line+"\t"+str(a[NNindex])+"\t"+str(b[NNindex])+"\t"+str(process[idx])
            line=line+"\t"+str(process[NNindex])+"\n"
            OUT.write(line)

    OUT.close()

#---
# PART 5: Creates density and cluster images.

img_cluster="PB-cluster" # use for output image filenames
img_density="PB-density" # use for output image filenames

from GD_util import *
GD_Maps(method,bitmap,Nprocess>window,nloop,height,width,img_cluster,img_density)

```

4.3 Home-made graphic library

On GitHub: [GD_util.py](#). Relies on the Pillow library, to perform various filtering processes directly in image bitmaps rather than on the initial data. The unsupervised clustering technique uses a color equalizer, as the main machine learning algorithm. The library has only one function GD_Maps that does both supervised classification at once.

```

import math
from PIL import Image, ImageDraw # ImageDraw to draw rectangles etc.

def GD_Maps(method,bitmap,Nprocess>window,nloop,height,width,img_cluster,img_density):

    #---
    # PART 1: Allocate first image (clustering), including colors (palette)

    img1 = Image.new( mode = "RGBA", size = (width, height), color = (0, 0, 0) )
    pix1 = img1.load() # pix[x,y]=col[n] to modify the RGB color of a pixel
    draw1 = ImageDraw.Draw(img1,"RGBA")

    coll=[]

```

```

coll.append((255,0,0,255))
coll.append((0,0,255,255))
coll.append((255,179,0,255))
coll.append((0,0,0,255))
coll.append((0,179,0,255))
for i in range(Nprocess,256):
    coll.append((255,255,255,255))
oldBitmap = [[255 for k in range(height)] for h in range(width)]
densityMap= [[0.0 for k in range(height)] for h in range(width)]
for pixelX in range(0,width):
    for pixelY in range(0,height):
        processID=bitmap[pixelX][pixelY]
        pix1[pixelX,pixelY]=coll[processID]
draw1.rectangle((0,0,width-1,height-1), outline = "black",width=1)
fname=img_cluster+'.png'
img1.save(fname)

#---
# PART 2: Filter bitmap and densityMap

for loop in range(nloop): #

    print("loop",loop,"out of",nloop)
    for pixelX in range(0,width):
        for pixelY in range(0,height):
            oldBitmap[pixelX][pixelY]=bitmap[pixelX][pixelY]

for pixelX in range(0,width):
    for pixelY in range(0,height):
        count=[0] * Nprocess
        density=0
        maxcount=0
        topProcessID=255 # dominant processID near (pixelX, pixelY)
        for u in range(-window>window+1):
            for v in range(-window>window+1):
                x=pixelX+u
                y=pixelY+v
                if x<0:
                    x+=width # boundary effect correction
                if y<0:
                    y+=height # boundary effect correction
                if x>=width:
                    x-=width # boundary effect correction
                if y>=height:
                    y-=height # boundary effect correction
                if method == 0:
                    dist2=1
                else:
                    dist2=1/math.sqrt(1+u*u + v*v)
                processID=oldBitmap[x][y]
                if processID < 255:
                    count[processID]=count[processID]+dist2
                    if count[processID]>maxcount:
                        maxcount=count[processID]
                        topProcessID=processID
                    density=density+dist2
            density=density/(10**loop) # 10 at power loop (dampening)
            densityMap[pixelX][pixelY]=densityMap[pixelX][pixelY]+density
            bitmap[pixelX][pixelY]=topProcessID

#---
# PART 3: Some pre-processing; output cluster image

densityCountHash={} # use to rebalance gray levels
for pixelX in range(0,width):
    for pixelY in range(0,height):

```

```

topProcessID=bitmap[pixelX][pixelY]
density=densityMap[pixelX][pixelY]
if density in densityCountHash:
    densityCountHash[density]=densityCountHash[density]+1
else:
    densityCountHash[density]=1
pix1[pixelX,pixelY]=col1[topProcessID]

draw1.rectangle((0,0,width-1,height-1), outline ="black",width=1)
fname=img_cluster+str(loop)+'.png'
img1.save(fname)

#---
# PART 4: Equalize gray levels in the density image; output image as a PNG file
# Also try https://www.geeksforgeeks.org/python-pil-imageops-equalize-method/

densityColorHash={}
col2=[]
size=len(densityCountHash) # number of elements in hash
counter=0

for density in sorted(densityCountHash):
    counter=counter+1
    quant=counter/size # always between zero and one
    if quant < 0.08:
        densityColorHash[density]=0
    elif quant < 0.18:
        densityColorHash[density]=30
    elif quant < 0.28:
        densityColorHash[density]=55
    elif quant < 0.42:
        densityColorHash[density]=90
    elif quant < 0.62:
        densityColorHash[density]=120
    elif quant < 0.80:
        densityColorHash[density]=140
    elif quant < 0.95:
        densityColorHash[density]=170
    else:
        densityColorHash[density]=254

# allocate second image (density image)

img2 = Image.new( mode = "RGBA", size = (width, height), color = (0, 0, 0) )
pix2 = img2.load() # pix[x,y]=col[n] to modify the RGB color of a pixel
draw2 = ImageDraw.Draw(img2,"RGBA")

# allocate gray levels (palette)
for i in range(0,256):
    col2.append((255-i,255-i,255-i,255))

# create density image pixel by pixel
for pixelX in range(0,width):
    for pixelY in range(0,height):
        density=densityMap[pixelX][pixelY]
        color=densityColorHash[density]
        pix2[pixelX,pixelY]=col2[color]

# output density image
draw2.rectangle((0,0,width-1,height-1), outline ="black",width=1)
fname=img_density+str(loop)+'.png'
img2.save(fname)

return()

```

References

- [1] Vincent Granville. The art of visualizing high dimensional data. *Preprint*, pages 1–17, 2022. MLTechniques.com [\[Link\]](#). 8
- [2] Vincent Granville. *Stochastic Processes and Simulations: A Machine Learning Perspective*. MLTechniques.com, 2022. [\[Link\]](#). 2, 3, 7
- [3] Vincent Granville, Mirko Krivanek, and Jean-Paul Rasson. Simulated annealing: A proof of convergence. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16:652–656, 1996. 4
- [4] Luuk Spreuwers. *Image Filtering with Neural Networks: Applications and Performance Evaluation*. PhD thesis, University of Twente, 1992. 6